Editor: **Robert Blumen**
SalesForce
robert@robertblumen.com

Justin Richer on OAuth

Gavin Henry

From the Editor
In Episode 376 of "Software Engineering Radio," Justin Richer, lead author of *OAuth2 in Action* and editor of OAuth extensions RFC 7591, 7592, and 7662, discusses the key technical features of the OAuth 2.0 protocol for authorization. Gavin Henry spoke with Richer about browser-based OAuth2, types of tokens, OpenID Connect, PKCE, JavaScript Object Notation Web Token pros and cons, where to store them, client secrets, single-page apps, mobile apps, current best practices, OAuth.XYZ, HEART, MITREid, token validation, dynamic client registration, the decision factors of the various types of authorization grants to use, and what is next for OAuth. To hear the full interview, visit http://www.se-radio.net or access our archives via RSS at http://feeds.feedburner.com/se-radio.—*Robert Blumen*

Gavin Henry: What is OAuth, and what is the difference between versions 1 and 2?

Justin Richer: It's a delegation protocol that allows you to delegate rights to another user to use an API [application programming interface]. You can set a service-specific password in an automated fashion. Instead of username and password, your application can get an OAuth access token to access the API for you. The application never needs to know who you are.

OAuth1 and 2 are conceptually similar; they differ in how they allow software to act on a user's behalf and the assumptions underlying their design. OAuth1's purpose was to connect two websites. As the Internet changed, people were building API-driven websites, and mobile applications interacted more on the Web, we took the best parts of OAuth1 and streamlined it. OAuth1 was a monolithic protocol; OAuth2 allows greater flexibility.

OAuth was a reaction to Web APIs that were deployed with HTTP basic authentication, asking for username and password. The OAuth token-based model allows you to create a new credential that represents just that piece of delegated software working for that user instead of that user and all of their rights to the protected API. A key strength of the OAuth token-based model is provision of a new token for every service and client application. They don't have to be memorized or managed by the user. Because everyone has a different identifiable access token, both the client and the server can monitor who's doing what.

How do mobile-app developers use OAuth2?

The app lives on the device, has access to the storage, and is saving a token or some type of data on that device. It runs, lives, and executes on the device itself, outside of the context of the system browser.

People weren't doing that when we were writing OAuth2. There was a brighter line between Web-based

Digital Object Identifier 10.1109/MS.2019.2949648
Date of current version: 24 December 2019

applications, browser-based applications, and native applications. JavaScript now runs everywhere, and there are wrappers, repackagers, and cross-compilers for many languages and platforms, so you can't predict how an application will function based on just the language that it's written in.

With mobile apps, or native apps in general, you can do a dynamic registration instead of a static registration. The instance of the client-application software wakes up and knows that to talk to this API, it needs to talk to a specific OAuth authorization server. But it will realize that it doesn't have a client identifier or client secrets or keys associated with it, so it must go get something.

It would know the endpoint at a minimum. Even then, there are extensions to OAuth that allow for resource-based discovery for common APIs, such as OpenID Connect. You can discover all of that from a single piece of user input. Once you know which authorization server you need to talk to, the client software directly calls the server. The server then can provide a client identifier, most importantly, and, in many cases, a client's secret.

If you have a mobile application installed on a million different devices, a client's secret is not secret anymore; a configuration-time secret that is part of the build or deployment process will not stay secret. But if we can turn that into a runtime secret, mobile apps do fine. A lot can be done to store and manage runtime secrets. There are even security enclaves covered by localized biometrics where you can store things on today's platforms.

How do you prove that that client is something you know about, to allow access?

There are some things built into the dynamic client-registration protocol in OAuth, such as RFC [request for comments] 7591. It is for native apps, but Web-server-based and browser-based apps can use it too. This protocol lets you set up the environment so that all of those assumptions about which secrets and identifiers are in place before the OAuth protocol begins can be put there.

At initial registration, you can pass on a software statement, which is a signed assertion, to help identify you. It's actually a JSON [JavaScript Object Notation] Web Token (JWT) format that identifies attributes the client should be asking for. This is like a fingerprint of this application, signed by a third party's trusted key. This allows us to look at a piece of software and anticipate expected behavior, so that we can recognize anomalies.

Are JWTs the preferred format for the tokens?

Not necessarily. JWTs are the most common structured, self-contained token format in OAuth, but neither OAuth1 nor OAuth2 care what the format of this token is; you just have to be able to put it into an HTTP header.

OpenID Connect was built as an identity layer on top of OAuth2. When OpenID Connect was being ratified, we looked at classic Open ID systems, which were a redirect-based, website-focused identity protocol. It was difficult to integrate them with OAuth1, which was a website-focused authorization protocol.

Nobody ever wants pure authentication or pure authorization. They almost always want to know something about the user so they can contact the user if something goes wrong. But when we were developing OAuth2, the idea with OpenID Connect was to start with the authorization protocol of OAuth2. You have the user show up and authorize access to something. But the question is, what are they authorizing access to? With OpenID Connect, they are authorizing access to their identity information. You treat their identity, instead of the authentication, as the primary thing—I have to know who you are, and then I'll figure out what I'm going to let you do.

As a consequence, OpenID Connect and similar identity protocols are the most common uses of OAuth2. Unfortunately, that means that people think of OAuth2 as an authentication protocol, or as a login system, which it is not.

What is one thing you wish you could teach every developer about authorization?

Authorization looks simple, but there are many places where it can go sideways. Read the best practices and use trusted libraries. OAuth has been around long enough that there is good, well-established community work available on any platform. Also, be aware that with single-page apps, OAuth and authorization delegation might not be the solution that your problem needs. OAuth is a powerful and elegant tool for doing one specific thing, but it's not a panacea. You can't just add it to a system and then magically get security.

ABOUT THE AUTHOR

GAVIN HENRY is the founder of SureVoIP, an Internet telephony service provider, and has written most of the software that sticks it all together. His research interests include all aspects of software engineering, identity management (especially around Open LDAP), all layers of the network stack, systems programming, and the free software ecosystem. Henry received his engineering degree in electronics communications. Contact him at ghenry@surevoip.co.uk.

IEEE Software (ISSN 0740-7459) is published bimonthly by the IEEE Computer Society. IEEE headquarters: Three Park Ave., 17th Floor, New York, NY 10016-5997. IEEE Computer Society Publications Office: 10662 Los Vaqueros Cir., Los Alamitos, CA 90720; +1 714 821 8380; fax +1 714 821 4010. IEEE Computer Society headquarters: 2001 L St., Ste. 700, Washington, DC 20036. Subscribe to *IEEE Software* by visiting www.computer.org/software.

Postmaster: Send undelivered copies and address changes to *IEEE Software*, Membership Processing Dept., IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854-4141. Periodicals Postage Paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Agreement Number 40013885. Return undeliverable Canadian addresses to PO Box 122, Niagara Falls, ON L2E 6S8, Canada. Printed in the USA.